

Clase práctica: Smalltalk

Paradigmas (de Lenguajes) de Programación

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

19 de noviembre de 2024

Clase de hoy

- 1 Repaso de los conceptos de P.O.O. y Smalltalk.
- 2 Métodos. Seguimiento.
- 3 Métodos de clase
- 4 Bloques.
- 5 Ejercicios Bloques.
- 6 Colecciones.
- 7 Booleanos.

Repaso del paradigma que utilizaremos hoy

¿Qué es un programa?

- Software como modelo computable de la realidad para comprender y resolver problemas.
- Cómputo basado en objetos enviándose **mensajes**.
- Evitar soluciones procedurales **delegando responsabilidades** en objetos.

El conjunto de **mensajes** que los objetos saben responder define su comportamiento y sus responsabilidades (es el QUÉ un objeto).

Tipos de mensajes

- **Unarios:** e.g., 2 pesos.
- **Binarios:** e.g., December first, 1985.^a
- **Keyword:** e.g., 'hola mundo' indexOf: \$o startingAt: 6.

^anotar que first es unario, ¿dónde está el binario?.

Objetos distintos para el 'qué' y el 'cómo'

¿Qué saben hacer los objetos?

Colaborar entre sí mediante el envío de mensajes. A esta secuencia de colaboraciones la llamamos **método**, y definen el **CÓMO**.

Observaciones

- Los métodos son objetos.
- La ejecución de un método también es un objeto.
- Ambos se pueden inspeccionar dentro del entorno de programación.

Comunicación entre objetos en Smalltalk

- Dirigida: Hay un emisor y un receptor.
- Sincrónica: Se espera a la respuesta del mensaje.
- Siempre hay respuesta: Si no se explicita, se retorna `self`.
- El receptor no conoce al emisor: siempre responde igual sin importar el emisor (salvo que el emisor se envíe como colaborador del mensaje).

Machete

Sintaxis

- “Comentarios”
- | var1 var2 ...|
- [:arg1 :arg2 | | var1 var2 | expresión]
- expresión1. expresión2. expresión3
- objeto mensaje
- objeto msj1; msj2
- var := expresión
- ^ expresión

Literales

- 123
- 123.4
- \$c
- 'texto'
- #símbolo
- #(123 123.3 \$a ábc' #abc)

Palabras Reservadas

- self
- super
- nil
- true
- false
- thisContext

Conociendo Pharo

Algunas herramientas básicas:

- **Playground/Workspace:** para interactuar con el sistema.
- **Transcript:** para registrar lo que pasa.
- **System browser:** para navegar las clases definidas.
- **Inspector:** para inspeccionar un objeto.
- **Debugger:** para ver si algo falla o ver en detalle la secuencia de mensajes.

Tip: Con `shift+enter` pueden navegar más rápido

Ejercicios Integer - Seguimiento

Implementar

- Implementar el método `mcm`: `aNumber` en la clase `Integer` para poder calcular el mínimo común múltiplo entre dos números.

Recordar que el mismo se calcula cómo $mcm(a, b) = \frac{a*b}{gcd(a,b)}$.
Asumir que cuenta con el mensaje `gcd`: `aNumber` implementado.

Seguimiento

- Realizar un seguimiento de la expresión `6 mcm: 10` y hacer el diagrama de secuencia correspondiente.
- Con esa información, completar la siguiente tabla:

Mensaje	Receptor	Colaboradores	Resultado
<code>mcm:</code>	6	10	...
...
...

Métodos de clase

¿Qué ocurre cuando mandamos un mensaje a una clase?

- ¡Lo mismo que siempre!
- Las clases son objetos.
- Como todo objeto, tienen sus colaboradores internos y su clase.

Cómo funciona el `new`

Dada la siguiente implementación:

```
Person class >> newWithName: nombre
  instancia := (self new).
  instancia firstName: nombre.
  nInstancias := nInstancias + 1.
  ^ instancia.
```

¿Qué ocurre si ejecutamos la siguiente colaboración?

```
Person newWithName: 'roberto'
```


Closures

Permiten representar un conjunto de colaboraciones. En definitiva, es segmento de código al cual no me importa ponerle un nombre (y tiene algunas características más, que veremos luego).

Sintaxis

```
[ :x :y |  
  | v |  
  v := x.  
  v * x + y  
]
```

¿Bloque, Lambda o Closure?

- Bloque: término genérico, designa una *porción de código*.
- Expresión lambda: proveniente del mundo funcional (Lisp).
- Closure: bloque con un contexto capturado, que también es un objeto, ¡obviamente!

Closures: Seguimiento

Los closures se ligan al contexto de ejecución donde son creados.
Tanto las variables como el return.

Ejercicio

¿Qué retorna cada envío de #value en el siguiente código si ejecutamos
B new m2? ¿Qué devuelve?

```
A >> m1
| x y |
y := 0.
x := [y := y + 1].
^ x.
```

```
B >> m2
| a aBlock anotherBlock |
a := A new.
aBlock := a m1.
aBlock value.
aBlock value.
anotherBlock := a m1.
anotherBlock value.
^ aBlock value.
```

Closures: Ejercicios

Implementar los siguientes mensajes en donde corresponda:

- #curry
- #timesDo:

Ejemplos

```
|currificado nuevo|  
currificado := [ :x :res | x + res ] curry.  
nuevo := currificado value: 10.  
nuevo value: 2.  
La última expresión debe evaluarse a 12.
```

```
|count copy|  
count := 0.  
10 timesDo: [copy := count. count := count + 2].  
Al terminar, count debe valer 20, y copy debe ser 18.
```

Ejercicios Integer - Seguimiento

Implementar

- Implementar el método `fact` en donde corresponda para que los números sepan responder a este mensaje que obtiene el factorial del número.

Seguimiento

- Realizar un seguimiento de la expresión `3 fact` y hacer el diagrama de secuencia correspondiente.
- Con esa información, completar la siguiente tabla:

Mensaje	Receptor	Colaboradores	Resultado
fact	3
...
...

Colecciones

Algunas conocidas

- Bag (Multiconjunto)
- Set (Conjunto)
- Array (Arreglo)
- OrderedCollection (Lista)
- SortedCollection (Lista ordenada)
- Dictionary (Hash)

Los mensajes `#with: with: ...`

Forma de crear estas colecciones.

Ejemplo

```
Bag with: 1 with: 2 with: 4
```

```
#(1 2 4) = (Array with: 1 with: 2 with: 4)
```

```
Bag withAll: #(1 2 4)
```

Mensajes más comunes

- `add`: agrega un elemento.
- `at`: devuelve el elemento en una posición.
- `at:put`: agrega un elemento a una posición.
- `includes`: responde si un elemento pertenece o no.
- `includesKey`: responde si una clave pertenece o no.

Colecciones

Mensajes más comunes

- `do`: evalúa un bloque con cada elemento de la colección.
- `keysAndValuesDo`: evalúa un bloque con cada par clave-valor.
- `keysDo`: evalúa un bloque con cada clave.
- `select`: devuelve los elementos de una colección que cumplen un predicado (filter de funcional).
- `reject`: la negación del `select`:
- `collect`: devuelve una colección que es resultado de aplicarle un bloque a cada elemento de la colección original (map de funcional).
- `detect`: devuelve el primer elemento que cumple un predicado.
- `detect:ifNone`: como `detect`:, pero permite ejecutar un bloque si no se encuentra ningún elemento.
- `reduce`: toma un bloque de dos o más parámetros de entrada y hace fold de los elementos de izquierda a derecha (foldl de funcional).

Colecciones: Map

El mensaje #do:

La forma de iterar queda definida por la colección

#map:

Implementemos el siguiente método en la clase Collection:

```
map: aBlock
```

Al ejecutarse, retorna la colección resultante de aplicar ese bloque a cada elemento de la colección original.

Ejemplo: **res** debe contener 6, 7 y 9 luego de ejecutar lo siguiente

```
| s res |  
s := Set with: 1 with: 2 with: 4.  
res := s map: [ :x | x + 5 ].
```

- ¿Cómo se decide qué clase de colección usar para el resultado, si podría ser de cualquier tipo?
- ¿Cómo se logra acceder desde el bloque al resultado parcial?

Colecciones: Mínimo

#minimo:

Agregar a la clase Collection un método con la siguiente interfaz:

`minimo: aBlock`

- aBlock es un bloque con un parámetro de entrada cuya evaluación devuelve un número.
 - El método debe evaluar el bloque en todos los elementos de la colección receptora, y devolver el mínimo de todos los valores obtenidos.
 - Se asume que la colección receptora no está vacía.
-
- ¿Cómo inicializar un primer valor?
 - ¿Funciona para Set?

Colecciones: Mínimo

Posible solución

```
minimo: aBlock
  | minElement minValue |
  self do: [:each | | val |
    minValue ifNotNil: [
      (val := aBlock value: each) < minValue ifTrue: [
        minElement := each.
        minValue := val]]
    ifNil: ["first element"
      minElement := each.
      minValue := aBlock value: each].
  ].
^minElement
```

El mensaje collect:

¿Qué devuelven las siguientes colaboraciones?

- `#hola collect: [:x | Unicode toUppercase: x]`.
- `(Interval from: 1 to: 5) collect: [:x | x*2]`.

Pista: los símbolos e intervalos son inmutables.

Veámoslo en el entorno.

El mensaje species

Las clases *Interval* y *ByteSymbol* redefinen el método `species` para poder responder a `collect`:

```
Interval >> species
```

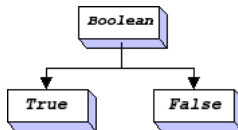
```
^Array.
```

```
ByteSymbol >> species
```

```
^ByteString.
```

¿Cómo se implementa el ifTrue?:

Recordar: *Boolean* tiene dos subclases.



True >> ifTrue: unBloque

^unBloque value.

False >> ifTrue: unBloque

^nil.

Otros métodos de *Boolean*

- `ifFalse:`
- `ifTrue:ifFalse:`
- `&`
- `|`
- `and:`
- `or:`
- `not`

¿Por qué los booleanos no entienden el mensaje `whileTrue:?`

¿Qué objetos lo entienden?

¿Dónde está implementado el método?